ORNL/TM-2008/###

ADIOS User's Manual

November 2008

Prepared by

C. Jin
S. Klasky
S. Hodson

Oak Ridge National Laboratory

J. Lofstead
F. Zheng
M. Wolf

Georgia Tech

R. Ross

Argonne National Laboratory

ORNL/TM-2008/###

ADIOS USER'S MANUAL

Prepared for the
Office of xxx
Xxxx Program
U.S. Department of Energy

C. Jin , S. Hodson, S. Klasky,
J. Lofstead, F. Zheng, M. Wolf, R. Ross

November 2008

Prepared by

OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6070
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

Figures

Abbreviations

ADIOS	Adaptive Input/Output System
API	***a****pplication* ***p****rogram* ***i****nterface*, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together. Most operating environments, such as MS-Windows, provide an API so that programmers can write applications consistent with the operating environment. Although APIs are designed for programmers, they are ultimately good for users because they guarantee that all programs using a common API will have similar interfaces. This makes it easier for users to learn new programs.
DART	Decoupled and Asynchronous Remote Transfers
GTC	Gyrokinetic Turbulence Code
HPC	high-performance computing
I/O	input/output
MDS	metadata server
MPI	Message-Passing Interface
NCCS	National Center for Computational Sciences
ORNL	Oak Ridge National Laboratory
OS	operating system
PG	process group
POSIX	Portable Operating System Interface
RDMA	remote direct memory access
XML	Extensible Markup Language

Acknowledgments

The Adaptive Input/Output (I/O) system (ADIOS) is a joint product of the National Center of Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) and the Center for Experimental Research in Computer Systems at the Georgia Institute of Technology. This work is being led by Scott Klasky (ORNL); Jay Lofstead (Georgia Tech, funded from Sandia Labs) is the main contributor. ADIOS has greatly benefited from the efforts of the following ORNL staff: Steve Hodson, who gave tremendous input and guidance; Chen Jin, who integrated ADIOS routines into multiple scientific applications; and Norbert Podhorszki, who integrated ADIOS with the Kepler workflow system. ADIOS also benefited from the efforts of the Georgia Tech team, including Prof. Karsten Schwan, Prof. Matt Wolf, Hassan Abbasi, and Fang Zheng. Wei Keng Liao, Northwestern University, and Wang Di, SUN, have also been invaluable in our coding efforts of ADIOS, writing several important . Essentially, ADIOS is componentization of I/O transport methods. Among the suite of transport methods, Decoupled and Asynchronous Remote Transfers (DART) was developed by Prof. Manish Parashar and his student Ciprian Docan of Rutgers University.

Without a scientific application, ADIOS would not have come this far. Special thanks go to Stephane Ethier at the Princeton Plasma Physics Laboratory (GTS); Researcher Yong Xiao and Prof. Zhihong Lin from the University of California, Irvine (GTC); Julian Cummings at the California Institute of Technology; and Seung-Hoe and Prof. C. S. Chang at New York University (XGC).

The manual was drafted by Chen Jin with substantive input from Steve Hodson, Scott Klasky, Jay Lofstead, Fang Zheng, Rob Ross, and Matt Wolf.

This project is sponsored by ORNL, Georgia Tech, The Scientific Data Management Center (SDM) at Lawrence Berkeley National Laboratory, and the U.S. Department of Defense.

ADIOS contributors

ANL: Rob Ross

Georgia Tech: Hasan Abbasi, Jay Lofstead, Karsten Schwan, Fang Zheng, Matthew Wolf

NCSU: Xiaosong Ma

Northwestern: Alok Choudhary, Wei Keng Liao

ORNL: Steve Hodson, Chen Jin, Scott Klasky, Norbert Podhorszki, Steve Poole

Rutgers: Ciprian Docan, Manish Parashar

SUN: Wang Di

Introduction

1.1 Goals

As computational power has increased dramatically with the increase in the number of processors, input/output (IO) performance has become one of the most significant bottlenecks in today's high-performance computing (HPC) applications. With this in mind, ORNL and the Georgia Institute of Technology's Center for Experimental Research in Computer Systems have teamed together to design the Adaptive I/O System (ADIOS) as a componentization of the IO layer, which is scalable, portable, and efficient on different clusters or supercomputer platforms. We are also providing easy-to-use, high-level application program interfaces (APIs) so that application scientists can easily adapt the ADIOS library and produce science without diving too deeply into computer configuration and skills.

1.2 What Is ADIOS?

ADIOS is a state-of-the-art componentization of the IO system that has demonstrated impressive IO performance results on leadership class machines and clusters; sometimes showing an improvement of more than 1000 times over well known parallel file formats. ADIOS is essentially an I/O componentization of different I/O transport methods. This feature allows flexibility for application scientists to adopt the best I/O method for different computer infrastructures with very little modification of their scientific applications. ADIOS has a suite of simple, easy-to-use APIs. Instead of being provided as the arguments of APIs, all the required metadata are stored in an external Extensible Markup Language (XML) configuration file, which is readable, editable, and portable for most machines.

1.3 The Basic ADIOS Group Concept

The ADIOS "group" is a concept in which input variables are tagged according to the functionality of their respective output files. For example, a common scientific application has checkpoint files prefixed with `restart` and monitoring files prefixed with `diagnostics`. In the XML configuration file, the user can define two separate groups with tag names of adios-group as "restart" and "diagnostic." Each group contains a set of variables and attributes that need to be written into their respective output files. Each group can choose to have different I/O transport methods, which can be optimal for their I/O patterns.

1.4 Other Interesting Features of ADIOS

ADIOS contains a new self-describing file format, BP. The BP file format was specifically designed to support delayed consistency, lightweight data characterization, and resilience. ADIOS also contains python scripts that allow users to easily write entire "groups" with the inclusion of one include statement inside their Fortran/C code. Another interesting feature of ADIOS is that it allow

users to use multiple I/O methods for a single group. This is especially useful if users want to write data out to the file system, simultaneously capturing the metadata in a database method, and visualizing with a visualization method.

1.5 ADIOS 2.0 Goals

One of the main goals for ADIOS 2.0 is to include subarray reads in the global domain and to produce faster reads via indexing methods. Another goal is to provide more advanced data types via XML in ADIOS so that it will be compatible with F90/c/C++ structures/objects. We will also be working on the IBM BlueGene P computer to provide full support for current and future architectures..

We will also work on the following advanced topics for ADIOS 2.0:

- A link to an external database for provenance recording.
- Autonomics through a feedback mechanism from the file system to optimize I/O performance. For instance, ADIOS can be adaptively changed from a synchronous to an asynchronous method or deciding when to write restart to improve I/O performance.
- A staging area for data querying, analysis, and in situ visualization.

Installation

1.6 Obtaining ADIOS

The ADIOS library can be checked out from the SVN repository, which requires a user name and a password. The check out command is:

```
svn co  https://svn.ccs.ornl.gov/svn-ewok/ADIOS/trunk
```

Or from the following website http://www.adios-api.org:

1.7 Quick Installation

To get started with ADIOS, the following steps can be used to configure, build, test, and install the ADIOS library, header files, and support programs.

```
cd trunk/

./runconf

make

make install
```

Note: ./runconf is the batch script containing a series of commands that set appropriate environment variables and configure options.

1.7.1 Linux cluster

The following is a snapshot of the batch scripts on Ewok, an Intel-based Infiniband cluster running Linux:

```
export CC=mpicc
./configure --preifix = <location for ADIOS software installation>
            --enable-dependency-tracking
            --with-mxml=<location of mini-xml installation>
            --with-hdf5=<location of HDF5 installation>
            --with-netcdf=<location of netCDF installation>
```

1.7.2 Cray XT4

To install ADIOS on a Cray XT4, the right compiler commands and configure flags need to be set. The required commands for ADIOS installation on Jaguar are as follows:

```
export CC=cc
export FC=ftn
./configure --preifix = <location for ADIOS software installation>
            --enable-dependency-tracking
            --with-mxml=<location of mini-xml installation>
            --with-hdf5=<location of HDF5 installation>
            --with-netcdf=<location of netCDF installation>
```

1.8 ADIOS Dependencies

1.8.1 Mini-xml parser (required)

The mini-xml library is used to parse XML configuration files. Therefore, the ADIOS library must have it defined in the configure flag. Otherwise, the library cannot be compiled

1.8.2 PHDF5 (optional)

If there is no HDF5 installed on the system, the offline serial version of the bp2h5 converter cannot be built and installed. If the Parallel HDF5 (PHDF5) library does not exist, PHDF5 will not be used as a transport method in ADIOS.

1.8.3 PnetCDF (optional)

In the same way as HDF5/PHDF5, the bp2ncd converter will not be built or installed if NetCDF installation path is not provided in the configure options.

1.8.4 MPI and MPI-IO (required)

Currently, most large-scale scientific applications rely on the Message Passing Interface (MPI) library to implement communication among processes. For instance, when the Portable Operating System Interface (POSIX) is used as transport method, the rank of each processor in the same communication group, which needs to be retrieved by the certain MPI APIs, is commonly used in defining the output files.

MPI-IO can also be considered the most generic I/O library on large-scale platforms. It is very difficult to find any platform without MPI or MPI-IO installed. Therefore, MPI and MPI-IO is required for the ADIOS 1.0 release.

1.8.5 Serial HDF5 and NetCDF (optional)

The HDF5 library and the bp2ncd converter need to be installed to build the serial version bp2h5 converter. Otherwise, the converter will not be built. We will continue to work on the parallelized converters, which will require PHDF5 and PnetCDF also.

1.9 Full Installation

The following list is the complete set of options that can be used with configure to build ADIOS and its support utilities:

```
--help              print the usage of ./configure command
--with-tags[=TAGS]  include additional configurations [automatic]
--with-mxml=DIR     Location of Mini-XML library
--with-gengetopt=<path> Location of gengetopt
--with-hdf5=<location of HDF5 installation>
--with-hdf5-incdir=<location of HDF5 includes>
--with-hdf5-libdir=<location of HDF5 library>
--with-netcdf=<location of NetCDF installation>
--with-netcdf-incdir=<location of NetCDF includes>
--with-netcdf-libdir=<location of NetCDF library>
```

Some influential environment variables are lists below:

```
  CC         C compiler command
  CFLAGS     C compiler flags
  LDFLAGS    linker flags, e.g. -L<lib dir> if you have libraries in a
             nonstandard directory <lib dir>
  CPPFLAGS   C/C++ preprocessor flags, e.g. -I<include dir> if you
have
             headers in a nonstandard directory <include dir>
  CPP        C preprocessor
  CXX        C++ compiler command
  CXXFLAGS   C++ compiler flags
  FC         Fortran compiler command
  FCFLAGS    Fortran compiler flags
  CXXCPP     C++ preprocessor
  F77        Fortran 77 compiler command
  FFLAGS     Fortran 77 compiler flags
  MPICC      MPI C compiler command
  MPIFC      MPI Fortran compiler command
```

ADIOS User APIs

As mentioned earlier, ADIOS is comprised of two parts: the XML configuration file and APIs. In this section, we will explain the functionality of each API in detail and how they are applied in the program.

1.10 High-Level API Description

1.10.1 Introduction

ADIOS provides both Fortran and C routines. All ADIOS routines and constants in both C and Fortran begin with the prefix "adios_" to avoid name collisions. For the remainder of this section, only the C versions of ADIOS APIs are presented. The primary differences between the C and Fortran routines are as follows:

Error Codes are returned in a separate argument for Fortran as opposed to the return value for C routines.

A unique feature of ADIOS is group implementation, which is constituted by a list of variables and associated with individual transport methods. This flexibility allows the applications to make the best use of the file system according to its own different I/O patterns.

1.10.2 ADIOS-required functions

This section contains the basic functions needed to integrate ADIOS into scientific applications. ADIOS is a lightweight I/O library, and there are only seven required functions from which users can write scalable, portable programs with flexible I/O implementation on supported platforms:

adios_init—initialize ADIOS and load the configuration file
adios_open—open the group associated with the file
adios_group_size—pass the group size to allocate the memory
adios_write—write the data either to internal buffer or disk
adios_read—associate the buffer space for data read into
adios_close—commit write/read operation and close the data
adios_finalize—terminate ADIOS

You can add functions to your working knowledge incrementally without having to learn everything at once. For example, you can achieve better I/O performance on some platforms by simply adding the asynchronous functions adios_start_calculation, adios_end_calculation, and adios_end_iteration to your repertoire. These functions will be detailed below in addition to the seven indispensable functions.

The following provides the detailed descriptions of required APIs when users apply ADIOS in the Fortran or C applications.

1.10.2.1 adios_init

This API is required only once in the program. It loads XML configuration file and establishes the execution environment. Before any ADIOS operation starts, `adios_init` is required to be called to create internal representations of various data types and to define the transport methods used for writing.

int adios_init (const char * `xml_fname`)
Input:
`xml_fname` – string containing the name of the XML configuration file

1.10.2.2 adios_open

This API is called whenever a new output file is opened. Adios_open, corresponding to `fopen` (not surprisingly), opens an adios-group given by `group_name` and associates it with one or a list of transport methods, which can be identified in future operations by the File structure whose pointer is returned as `fd_p`. The group name should match the one defined in the XML file. The I/O handle `fd_p` prepares the data types for the subsequent calls to write data using the io_handle. The third argument , `file_name`, is a string representing the name of the file. As the last argument, `mode` is a string containing a file access mode. It can be any of these three mode specifiers: "`r`," "`w`," or "`a`." Currently, ADIOS supports three access modes: "write or create if file does not exist," "read," and "append file." The call opens the file only if no coordination is needed among processes for transport methods that the users have chosen for this adios_group, such as POSIX method. Otherwise, the actual file will be opened in adios_group_size based on the provided argument `comm`, which will be examined in Sect. 4.1.2.3.

int adios_open (int64_t * `fd_p`, const char * `group_name`
,const char * `file_name`, const char * `mode`)
Input:
`fd_p`—pointer to the internal file structure
`group_name`—string containing the name of the group
`file_name`—string containing the name of the file to be opened
`mode`—string containing a file access mode.

1.10.2.3 adios_group_size

This function passes the size of the group to the internal ADIOS transport structure to facilitate the internal buffer management and to construct the group index table. The first argument is the file handle. The second argument is the size of the payload for the group opened in the adios_open routine. This value can be calculated manually or through our python script. It does not affect read operation because the size of the data can be retrieved from the file itself. The third argument is the returned value for the total size of this group, including payload size and the metadata overhead. The value can be used for performance benchmarks, such as I/O speed. As the last argument, we pass the pointer of

coordination communicator down to the transport method layer in ADIOS. This communicator is required in MPI-IO–based methods such as collective and independent MPI-IO.

int adios_group_size (int64_t * fd_p, uint64_t group_size, uint64_t * total_size, void * comm)
Input:
fd_p—pointer to the internal file structure
group_size—size of data payload in bytes to be written out. If there is an integer 2 × 3 array, the payload size is 4*2*3 (4 is the size of integer)
comm—communicator (handle) for multi-process coordination
output :
total_size—the total sum of payload and overhead, which includes name, data type, dimensions and other metadata)

1.10.2.4 adios_write

The adios_write routine submits a data element var for writing and associates it with the given var_name, which has been defined in the adios group opened by adios_open. If the ADIOS buffer is big enough to hold all the data that the adios group needs to write, this API only copies the data to buffer. Otherwise, adios_write will write to disk without buffering. Currently, adios_write supports only the address of the contiguous block of memory to be written. In the case of a noncontiguous array comprising a series of subcontiguous memory blocks, var should be given separately for each piece.

In the next XML section, we will further explain that var_name is the value of attribute "name" while var is the value of attribute "gwrite," both of which are defined in the corresponding <var> element inside adios_group in the XML file. By default, it will be the same as the value of attribute "name" if "gwrite" is not defined.

int adios_write (int64_t fd_p, const char * var_name, void * var)
Input:
fd_p—pointer to the internal file structure
var_name—string containing the annotation name of scalar or vector inthe file
var —the address of the data element defined need to be written

1.10.2.5 adios_read

Similar to adios_write, adios_read submits a buffer space var for reading a data element into. This does NOT actually perform the read. Actual population of the buffer space will happen on the call to adios_close. In other words, the value(s) of var can only be utilized after adios_close is performed. Here, var_name corresponds to the value of attribute "gread" in the <var> element declaration while var is mapped to the value of attribute "name." By default, it will be as same as the value of attribute "name" if "gread" is not defined.

int adios_read (int64_t `fd_p`, const char * `var_name`, uint64_t `read_size`,
void * `var`
)
Input:
`fd_p` – pointer to the internal file structure
`var_name` – the name of variable recorded in the file
`var` – the address of variable defined in source code
`read_size` – size in bytes of the data to be read in

1.10.2.6 adios_close

The adios_close routine commits the writing buffer to disk, closes the file, and releases the handle. At that point, all of the data that have been copied during adios_write will be sent as-is downstream. If the handle were opened for read, it would fetch the data, parse it, and populate it into the provided buffers. This is currently hard-coded to use posix I/O calls.

int adios_close (int64_t * `fd_p`);
Input:
`fd_p` – pointer to the internal file structure

1.10.2.7 adios_finalize

The `adios_finalize` routine releases all the resources allocated by ADIOS and guarantees that all remaining ADIOS operations are finished before the code exits. The ADIOS execution environment is terminated once the routine is fulfilled. The `proc_id` parameter provides users the opportunity to customize special operation on `proc_id`—usually the ID of the head node.

int adios_finalize (int `proc_id`)

Input:
`proc_id` – the rank of the processes in the communicator or the user-defined coordination variable

1.10.3 Nonblocking functions

1.10.3.1 adios_end_iteration

The adios_end_iteration provides the pacing indicator. Based on the entry in the XML file, it will tell the transport method how much time has elapsed in a transfer.

1.10.3.2 adios_start_ calculation/ adios_end_calculation

Together, adios_start_calculation and adios_end_calculation indicate to the scientific code when nonblocking methods should focus on engaging their I/O communication efforts because the process is mainly performing intense, stand-alone computation. Otherwise, the code is deemed likely to be communicating heavily for computation coordination. Any attempts to write or read during those

times will negatively impact both the asynchronous I/O performance and the interprocess messaging.

1.10.4 Other function

One of our design goals is to keep ADIOS APIs as simple as possible. In addition to the basic I/O functions, we provide another routine listed below.

1.10.4.1 adios_get_write_buffer

The adios_get_write_buffer function returns the buffer space allocated from the ADIOS buffer domain. In other words, instead of allocating memory from free memory space, users can directly use the allocated ADIOS buffer area and thus avoid copying memory from the ADIOS buffer to a user-defined buffer.

int adios_get_write_buffer (int64_t *fd_p*, const char * var_na*me*, uint64_t * *size*, void ** *buffer*)
Input:
fd_p – pointer to the internal File structure
`var_name` – name of the variable that will be read
size – size of the buffer to request
output:
buffer – initial address of read-in buffer for storing the data of `var_name`

1.10.5 Create a first ADIOS program

Figure 1 is a programming example that illustrates how to write a double-precision array `t` and a double-precision array with size of `NX` into file called "test.bp," which is organized in BP, our native tagged binary file format. This format allows users to include rich metadata associated with the block of binary data as well the indexing mechanism for different blocks of data (see Chap. 5).

```
/*example of parallel MPI write into a single file */

#include <stdio.h>    // ADIOS header file required
#include "adios.h"
int main (int argc, char *argv[])
{
    int i, rank, NX;
    double t [NX];
    // ADIOS variables declaration
    int64_t handle;
    uint_64 total_size;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Init ( &argc, &argv);
    MPI_Comm_rank (comm, &rank);
```

```
    // data initialization
    for ( i=0; i<NX; i++)
        t [i] = i * (rank+1) + 0.1;

    // ADIOS routines
    adios_init ( “config.xml”);
    adios_open (&handle, “temperature”, “data.bp”, “w”);
    adios_group_size (handle, 4, total_size, &comm);
    adios_write (handle, ”NX”, &NX);
    adios_write (handle, ”temperature”, t);
    adios_close (handle);
    adios_finalize (rank);

    MPI_Finalize();
    return 0;
}
```

Fig. 1. ADIOS programming example.

XML Config File Format

1.11 Overview

XML is designed to allow users store as much metadata as they can in an external configuration file. Thus the scientific applications are less polluted and require less effort to be verified again.

First, we present the XML template. Second, we demonstrate how to construct the XML file from the user's own source code. Third, we note how to troubleshoot and debug the errors in the file.

Abstracting metadata, data type, and dimensions from the source code into an XML file gives users more flexibility to annotate the arrays or variables and centralizes the description of all the data structures, which in return, allows I/O componentization for different implementation of transport methods. By cataloguing the data types externally, we have an additional documentation source as well as a way to easily validate the write calls compared with the read calls without having to decipher the data reorganization or selection code that may be interspersed with the write calls. It is useful that the XML name attributes are just strings. The only restrictions for their content are that if the item is to be used in a dataset dimension, it must not contain commas and must contain at least one non-numeric character. This is useful for incorporating expressions as various array dimensions elements. Figure 2 illustrates the corresponding XML configuration for the example we demonstrated in Fig. 1.

At a minimum, a configuration document must declare an `adios-config` element. It serves as a container for other elements; as such, it MUST be used as the root element. The expected children in any order would be `adios-group`, method, and buffer. The main elements of the xml file format are of the format

<element-name attr1 attr2 …>

```
<adios-config>
    <adios-group>
        <var />
        ……
        <attribute />
        ……
    </adios-group>
    …
    <method>
    ……
    <buffer>
</adios-config>
```

Fig. 2. Example XML configuration.

1.12 adios-group

The adios-group element represents a container for a list of variables that share the common I/O pattern as stated in the basic concepts of ADIOS in first chapter. In this case, the group domain division logically corresponds to the different functions of output in scientific applications, such as restart, diagnosis, and snapshot. Depending on the different applications, adios-group can occur as many times as is needed.

1.12.1 Declaration

The following example illustrates how to declare an adios group inside an XML file. First we start with adios-group as our tag name, which is case insensitive. It has an indispensable attribute called "name," whose value is usually defined as a descriptive string indicating the function of the group. In this case, the string is called "restart," because the files into which this group is written are used as checkpoints. The second attribute "host-language" indicates the language in which this group's I/O operations are written. The value of attribute "coordination-communicator" is used to coordinate the operations on a shared file accessed by a multiple process in the same communicator domain. "Coordination-var" provides the ability to use the user-defined variable, for example mype, rather than an MPI communicator for file coordination.

<**adios-group name**="restart"
host-language="C"
coordination-communicator="comm"
coordination-var="mype"
time-index="iter"/>

Required:

- name—containing a descriptive string to name the group

Optional:

- host-language—language in which the source code for group is written
- coordination-communicator—MPI-IO writing to a shared file
- coordination-var—coordination variables for non-MPI methods, such as Datatap method
- time-index—

1.12.2 Variables

The nested variable element "var" for adios_group, which can be either an array or a primitive data type, is determined by the dimension attribute provided.

1.12.2.1 Declaration

The following is an example showing how to define a variable in the XML file.

<**var name**="z-plane ion particles"
gwrite="zion"
gread="zion_read"
type="adios_real"
dimensions="7,mimax"
read="yes"/>

1.12.2.2 Attribute list

The attributes associated with var element as follows:

Required:

- name – the string name of variable stored in the output file
- type – the data type of the variable

Optional:

- gwrite – the value will be used in the python scripts to generate adios_write routines; the default value will be the same as attribute *name* if gwrite is not defined.
- gread – the value will be used in the python scripts to generate adios_read routines' the default value will be the same as attribute *name* if gread is not defined.
- path - HDF-5-style path for the element or path to the HDF-5 group or data item to which this attribute is attached. The default value is "/".
- dimensions - a comma-separated list of numbers and/or names that correspond to integer var elements determine the size of this item. If not specified, the variable is scalar.
- read – value is either *yes* or *no*; in the case of no, the adios_read routine will not be generated for this var entry. If undefined, the default value will be yes.

1.12.3 Attributes

The attribute element for adios_group provides the users with the ability to specify more descriptive information about the variables or group. The attributes can be defined in both static or dynamic fashions.

1.12.3.1 Declaration

The static type of attributes can be defined as follows:

<**attribute** name="experimental date"
path="/zion"
value="Sep-19-2008"

type=”adios_real”/>

If an attribute has dynamic value that is determined by the runtime execution of the program, it can be specified as follows:
<**attribute** name=“experimental date”
path=“/zion”
var=”time”/>

where var “time” need to be defined in the same adios-group.

1.12.3.2 Attribute list

Required:

- name - name of the attribute
- path – hierarchical path inside the file for the attribute
- value – attribute has static value of the attribute, mutually exclusive with the attribute *var*
- type – string or numeric type, paired with attribute *value*, in other words,, mutually exclusive with the attribute *var* also
- var – attribute has dynamic value that is defined as a variable in *var*

1.12.4 Gwrite/src

1.12.4.1 The element <Gwrite/src> is unlike <var> or <attribute>, which are parsed and stored in the internal file structure in ADIOS. The element <gwrite> only affects the execution of python scripts (see Chap. 10). Any content (usually comments, conditional statements, or loop statements) in the value of attribute “src” is copied identically into generated pre-processing files. Declaration

<gwrite src=” ”/>

Required:

- src - any statement that needs to be added into the source code. This code must will be inserted into the source code, and must be able to be compiled in the host language, C or Fortran.

1.12.5 Global arrays

Global-bounds is an optional nested element for adios-group. It specifies the global space and offsets within that space for the enclosed variable elements. In the case of writing to a shared file, the global-bounds information is recorded in BP file and can be interpreted by converters or other postprocessing tools or used to write out either HDF5 or NetCDF files by using PHDF5 or the PnetCDF method.

1.12.6 Time-index

ADIOS allows a dataset to be expanded in the space domain given by global bounds and in time domain. It is very common for scientific applications to write out a monitoring file at regular intervals. The file usually contains a group of time-based variables that have undetermined dimensional value on the time axis. ADIOS is Similar to NetCDF in that it accumulates the time-index in terms of the number of records, which theoretically can be added to infinity.

If any of variables in an adios group are time based, they can be marked out by adding the time-index variable as another dimension value.

1.12.6.1 Declaration

<**global-bounds dimensions**="nx_g, ny_g"
offsets="nx_o,0"/>
</**global-bounds**>

Required:

- dimensions - the dimension of global space
- offsets – the offset of the data set in global space

Any variables used in the global-bounds element for dimensions or offsets declaration need to be defined in the same adios-group as either variables or attributes.

For detailed global arrays use, see the examples illustrated in Sects. 10.7., Transport Methods.

Changing I/O Without Changing Source: The method element provides the hook between the adios-group and the transport methods. The user employs a different transport method simply by changing the method attribute of the method element. If more than one method element is provided for a given group, each element will be invoked in the order specified. This neatly gives triggering opportunities for workflows. To trigger a workflow once the analysis data set has been written to disk, the user makes two element entries for the analysis adios-group. The first indicates how to write to disk, and the second performs the trigger for the workflow system. No recompilation, relinking, or any other code changes are required for any of these changes to the XML file.

1.12.7 Declaration

The transport element is used to specify the mapping of an I/O transport method, including optional initialization parameters, to the respective adios-group. There are two major attributes required for the method element:

<**transport group**="restart"
method="MPI"
priority="1"

iteration="100"/>

Required:

- group - corresponds to an adios-group specified earlier in the file.
- method – a string indicating a transport method to use with the associated adios-group

Optional:

- priority– a numeric priority for the I/O method to better schedule this write with others that may be pending currently
- base-path–the root directory to use when writing to disk or similar purposes
- iterations– a number of iterations between writes of this group used to gauge how quickly this data should be evacuated from the compute node

1.12.8 Methods list

As the componentization of the IO substrate, ADIOS supports a list of transport methods, described in Section 0:

- NULL
- POSIX
- POSIXN2M (soon)
- MPI-IO
- MPI-CIO (soon)
- PHDF5
- PNetCDF (soon)
- MPI-AIO (soon)
- DATATAP (soon)

1.13 Buffer specification

The buffer element defines the attributes for internal buffer size and creating time that apply to the whole application (Fig. 3). The attribute allocate-time is identified as being either "now" or "oncall" to indicate when the buffer should be allocated. An "oncall" attribute waits until the programmer decides that all memory needed for calculation has been allocated. It then calls upon ADIOS to allocate buffer. There are two alternative attributes for users to define the buffer size: MB and free-memory-percentage.

1.13.1 Declaration

<**buffer size-MB**="100"
allocate-time="now" />

Required:

- size-MB – the user-defined size of buffer in megabytes. ADIOS can at most allocate from compute nodes. It is exclusive with free-memory-percentage.
- free-memory percentage – the user-defined percentage from 0 to 100% of freememory available on the machine. It is exclusive with size-MB.
- allocate-time – indicates when the buffer should be allocated

1.14 An Example XML file

```
<adios-config host-language="C">

  <adios-group name="temperature" coordination-communicator="comm">
    <var name="NX" type="integer"/>
    <var name="t" type="double" dimensions="NX"/>
    <attribute name="recorded date" path="/" value="Sep 19, 2008" type="string"/>
  </adios-group>

  <method group=" temperature " method="MPI"/>

  <buffer size-MB="1" allocate-time="now"/>

</adios-config>
```

Fig. 3. Example XML file for time allocation.

Transport methods

Because of the time it can take to move data from one process to another or to write and read data to and from a disk, it is often advantageous to arrange the program so that some work can be done while the messages are in transit. So far, we have used non-blocking operations to avoid waiting. Here we describe some details for arranging a program so that computation and I/O can take place simultaneously.

1.15 Synchronous methods

1.15.1 NULL

The ADIOS NULL method allows users to quickly comment out an ADIOS group. by changing the transport method to "NULL," users can test the speed of the routine by timing the output against no I/O. This is especially useful when working with asynchronous methods, which take an indeterminate amount of time. Another useful feature of this I/O is that it quickly allows users to test out the system and determine whether bugs are caused by the I/O system or by other places in the codes.

1.15.2 POSIX

The simplest method provided in ADIOS just does binary POSIX I/O operations. Currently, it does not support shared file writing or reading and has limited additional functionality. The main purpose for the POSIX I/O method is to provide a simple way to migrate a one-file-per-process I/O routine to ADIOS and to test the results without introducing any complexity from MPI-IO or other I/O methods. Performance gains just by using this transport method are likely due to our aggressive buffering for better streaming performance to storage. The buffering method writes out files in BP format, which is a compact, self-describing format.

Additional features may be added to the ADIOS POSIX transport method over time. a new transport method with a related name, such as POSIX-ASCII, may be provided to perform I/O with additional features. The POSIX-ASCII example would write out a text version of the data formatted nicely according to some parameters provided in the XML file.

1.15.3 MPI-IO

Many large-scale scientific simulations generate a large amount of data, spanning thousands of files or datasets. The use of MPI-IO reduces the amount of files and thus is helpful for data management, storage, and access.

The original MPI-IO method was developed based on our experiments with generating the better MPI-IO performance on the ORNL Jaguar machine. Many of his insights have helped us achieve excellent performance on both the Jaguar XT4 machine and on the other clusters. Some of the key insights we have taken advantage of include artificially serialized MPI_File_open calls and additional

timing delays that can achieve reduced delays due to metadata server (MDS) conflicts on the attached Lustre storage system.

The adapted code takes full advantage of NxM grouping through the coordination-communicator. This grouping generates one file per coordination-communicator with the data stored sequentially based on the process rank within the communicator. Figure 4 presents in the example of GTC code, 32 processes in the same Toroidal zone write to one integrated file. Additional serialization of the MPI_File_open calls is done using this communicator as well because each process may have a different size data payload. Rank 0 calculates the size that it will write, calls MPI_File_open, and then sends its size to rank 1. Rank 1 listens for the offset to start from, adds its calculated size, does an MPI_File_open, and sends the new offset to rank 2. This continues for all processes within the communicator. Additional delays for performance based on the number of processes in the communicator and the projected load on the Lustre MDS can be used to introduce some additional artificial delays that ultimately reduce the amount of time the MPI_File_open calls take by reducing the bottleneck at the MDS. An important fact to be noted is that individual file pointers are retrieved by MPI_File_open so that each process has its own file pointer for file seek and other I/O operations.

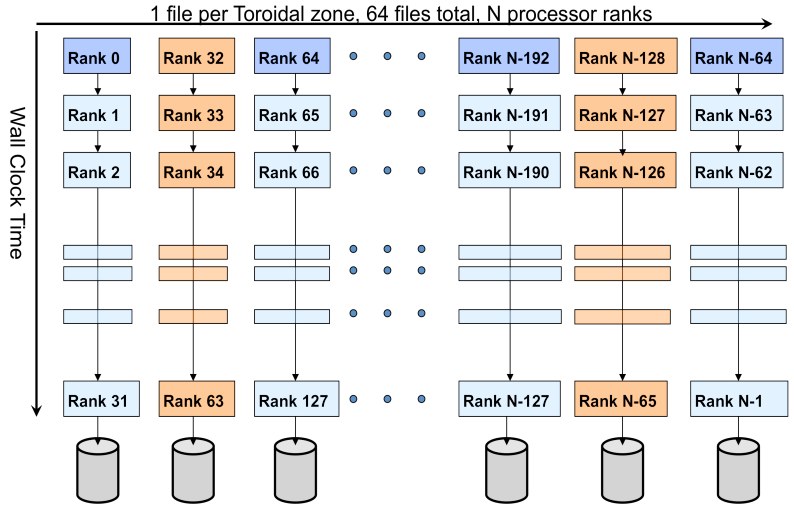

Fig. 4. Server-friendly metadata approach: offset the create/open in time.

We built the MPI-IO transport method, mainly with Lustre in mind because it is the primary parallel storage service we have available. However, other file -system -specific tunings are certainly possible and fully planned as part of this transport method system. For each new file system we encounter, a new transport method implementation tuned for that file system, and potentially that platform, can be developed without impacting any of the scientific code.

The MPI-IO transport method for Lustre is the most mature, fully featured, and well tested. We recommend to anyone creating a new transport method that they

study it as a model of full functionality and some of the advantages that can be made through careful management of the storage resources.

1.15.4 MPI-CIO

MPI-IO defines a set of portable programming interfaces that enable multiple processes to have concurrent access to shared files [1]. It is often used to store and retrieve structured data in their canonical order. The interfaces are split into two types: collective I/O and independent I/O. Collective functions require all processes to participate. Independent I/O, in contrast, requires no process synchronization.

Collective I/O enables process collaboration to rearrange I/O requests for better performance [2,3]. The collective I/O method in ADIOS first defines MPI fileviews for all processes based on the data partitioning information provided in the XML configuration file. ADIOS also generates MPI-IO hints, such as data sieving and I/O aggregators, based on the access pattern and underlying file system configuration. The hints are supplied to the MPI-IO library for further performance enhancement. The syntax to describe the data-partitioning pattern in the XML file uses the <global-bounds dimensions offsets> tag, which defines the global array size and the offsets of local subarrays in the global space.

The global-bounds element contains one or more nested var elements, each specifying a local array that exists within the described dimensions and offset. Multiple global-bounds elements are permitted, and strictly local arrays can be specified outside the context of the global-bounds element.

As with other data elements, each of the attributes of the global-bounds element is provided by the adios_write call. The dimensions attribute is specified by all participating processes and defines how big the total global space is. This value must agree for all nodes. The offset attribute specifies the offset into this global space to which the local values are addressed. The actual size of the local element is specified in the nested var element(s). For example, if the global bounds dimension were 50 and the offset were 10, then the var(s) nested within the global-bounds would all be declared in a global array of 50 elements with each local array starting at an offset of 10 from the start of the array. If more than one var is nested within the global-bounds, they share the declaration of the bounds but are treated individually and independently for data storage purposes.

Currently this method is unfinished at the time of writing, but will be released in the next minor release in Q1 2009.

1.15.5 PHDF5

HDF5, as a hierarchical File structure, has been widely adopted for data storage in various scientific research fields. Parallel HDF5 (PHdF5) provides a series of APIs to perform the I/O operations in parallel from multiple processors, which dramatically improves the I/O performance of the sequential approach to read/write an HDF5 file. In order to make the difference in transport methods

and file formats transparent to the end users, we provide a mechanism that write/read an HDF5 file with the same schema by keeping the same common adios routines with only one entry change in the XML file. this method provides users with the capability to write out exactly the same HDF5 files as those generated by their original PHDF5 routines. Doing so allows for the same analysis tool chain to analyze the data.

Currently, HDF5 supports two I/O modes: independent and Collective read or write, which can use either the MPI or the POSIX driver by specifying the dataset transfer property list in H5Dwrite function calls. In this release, only the MPI driver is supported in ADIOS; later on, both I/O drivers will be supported by changing the attribute information for PHDF5 method elements in XML.

1.15.6 PNetCDF

Another widely accepted standard file format is called NetCDF, which is the most frequently used file format in the climate and weather research communities. In ADIOS 2.0, this method will be supported.

1.15.7 Other methods

ADIOS provides an easy plug-in mechanism for users or developers to design their own transport method. A step-by-step instruction for inserting a new I/O method is given in Sect. 14.2. Users are likely to choose the best method from among the supported or customized methods for the running their platforms, thus avoiding the need to verify their source codes due to the switching of I/O methods.

1.16 Asynchronous methods

1.16.1 MPI-AIO

The initial implementation of the asynchronous MPI-IO method (MPI-AIO) is patterned after the MPI-IO method. Scheduled metadata commands are performed with the same serialization of MPI_Open calls as given in Fig. 4.

The degree of I/O synchronicity depends on several factors. First, the ADIOS library must be built with versions of MPI that are built with asynchronous I/O support through the MPI_File_iwrite, MPI_File_iread, and MPI_Wait calls. If asynchronous I/O is not available, the calls revert to synchronous (read blocking) behavior identical to the MPI-IO method described in the previous section.

Another important factor is the amount of available ADIOS buffer space. In the MPI-IO method, data are transported and ADIOS buffer allocation is reclaimed for subsequent use with calls to adios_close (). In the MPI-AIO method, the “close” process can be deferred until buffer allocation is needed for new data. However, if the buffer allocation is exceeded, the data must be synchronously transported before the application can proceed.

The deferral of data transport is key to effectively scheduling asynchronous I/O with a computation (to be implemented in version 2.0). In ADIOS version 1.0, the

application explicitly signals that data transport must be complete with intelligent placement of the adios_close () call to indicate when I/O must be complete. Later versions of ADIOS will perform I/O between adios_begin_calculation and adios_end_calculation calls, and complete I/O on adios_end_iteration calls.

This method will be available during Q1 2009.

1.16.2 DataTap

DataTap is an asynchronous data transport method built to ensure very high levels of scalability through server-directed I/O [7,8]. It is implemented as a request-read service designed to bridge the order-of-magnitude difference between available memories on the I/O partition compared with the compute partition. We assume the existence of a large number of compute nodes producing data (we refer to them as "*DataTap* clients") and a smaller number of I/O nodes receiving the data (we refer to them as "*DataTap* servers") (see Fig. 5).

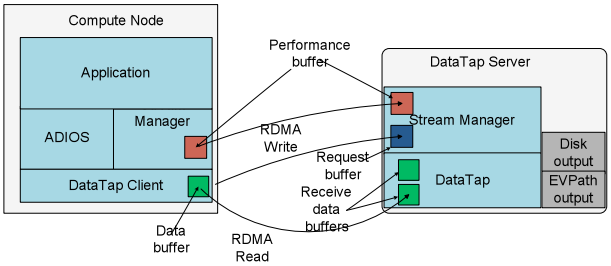

Fig. 5. DataTap architecture.

Upon application request, the compute node marks up the data in PBIO [9] format and issues a request for a data transfer to the server. The server queues the request until sufficient receive buffer space is available. The major cost associated with setting up the transfer is the cost of allocating the data buffer and copying the data. However, this overhead is small enough to have little impact on the overall application runtime. When the server has sufficient buffer space, a remote direct memory access (RDMA) read request is issued to the client to read the remote data into a local buffer. The data are then written out to disk or transmitted over the network as input for further processing in the I/O Graph.

We used the Gyrokinetic Turbulence Code (GTC) as an experimental tested for the DataTap transport. GTC is a particle-in-cell code for simulating fusion within tokamaks, and it is able to scale to multiple thousands of processors. In its default I/O pattern, the dominant I/O cost is from each processor writing out the local particle array into a file. Asynchronous I/O reduces this cost to just a local memory copy, thereby reducing the overhead of I/O in the application.

1.16.3 Decoupled and Asynchronous Remote Transfers (DART)

DART is an asynchronous I/O transfer method within ADIOS that enables low-overhead, high-throughput data extraction from a running simulation. DART consists of two main components: (1) a DARTClient module and (2) a

DARTServer module. Internally, the DART system uses RDMA to implement communication, coordination, and data transport between the DARTClient and the DARTServer modules.

The DARTClient module is a light library that implements the asynchronous I/O API. It integrates with the ADIOS layer by extending the generic ADIOS data transport hooks. It uses the ADIOS layer features to collect and encode the data written by the application into a local transport buffer. Once it has collected data from a simulation, DARTClient notifies the DARTServer through a coordination channel that it has data available to send out. DARTClient then returns and allows the application to continue its computations while data are asynchronously extracted by the DARTServer.

The DARTServer module is a stand-alone service that runs independently of the simulation. It typically runs on dedicated I/O nodes, and transfers data from the DARTClients and to remote sites (e.g., a remote a storage system such as the Luster file system. One instance of the DARTServer can service multiple DARTClients instances in parallel. Further, the server can run in cooperative mode (i.e., multiple instances of the server cooperate to service the clients in parallel and to balance load). The DARTServer receives notification messages from the clients, schedules the requests, and initiates the data transfers from the clients in parallel. The server schedules and prioritizes the data transfers while the simulation is computing in order to overlap data transfers with computations, to maximize data throughput, and to minimize the overhead on the simulation.

Currently this module will not be released in ADIOS 1.0.

BP file format

1.17 Introduction

This chapter describes the file structure of BP, which is the ADIOS native binary file format, to aid in understanding ADIOS performance issues and how files convert from BP files to other scientific file formats, such as netCDF and HDF5.

To avoid the file size limitation of 2 gigabytes by using a signed 32-bit offset within its internal structure, BP format uses an unsigned 64-bit datatype as the file offset. Therefore, it is possible to write BP files that exceed 2 gigabytes on platforms that have large file support.

By adapting ADIOS read routines based on the endianness indication in the file footer, BP files can be easily portable across different machines (e.g., between the Cray-XT4 and BlueGene).

To aid in data selection, we have a low-overhead concept of data characteristics to provide an efficient, inexpensive set of attributes that can be used to identify data sets without analyzing large data content.

As shown in Fig. 6, the BP format comprises a series of process groups and the file footer. The remainder of this chapter describes each component in detail and helps the user to better understand (1) why BP is a self -describing and metadata-rich file format and (2) why it can achieve high I/O performance on different machine infrastructures.

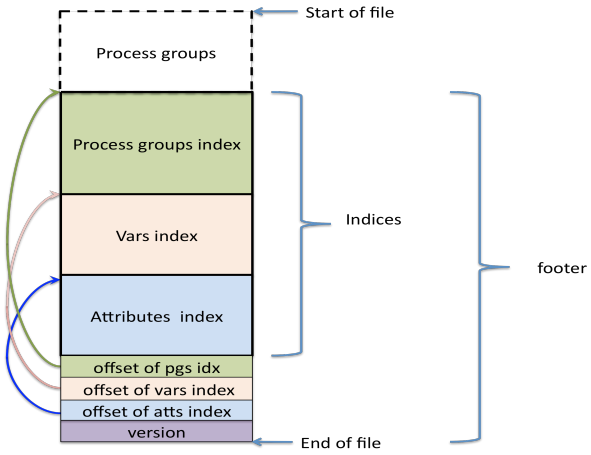

Fig. 6. BP file structure.

1.18 Footer

One known limitation of the NetCDF format is that the file contents are stored in a header that is exactly big enough for the information provided at file creation. Any changes to the length of that data will require moving data. To avoid this cost, we choose to employ a foot index instead. We place our version identifier

and the offset to the beginning of the index as the last few bytes of our file, making it simple to find the index information and to add new and different data to our files without affecting any data already written.

1.18.1 Version

We reserve 4 bytes for the file version, in which the highest bit indicates endianness. Because ADIOS uses a fixed-size type for data, there is no need to store type size information in the footer.

1.18.2 Offsets of indices

In BP format, we store three 8-byte file offsets right before the version word, which allows users or developers to quickly seek any of the index tables for process groups, variables, or attributes.

1.18.3 Indices

1.18.3.1 Characteristics

Before we dive into the structures of the three index tables mentioned earlier, let's first take a look what characteristic means in terms of BP file format. To be able to make a summary inspection of the data to determine whether it contains the feature of greatest interest, we developed the idea of data characteristics. The idea of data characteristics is to collect some simple statistical and/or analytical data during the output operation or later for use in identifying the desired data sets. Simple statistics like array minimum and maximum values can be collected nearly for free as part of the I/O operation. Other more complex analytical measures like standard deviations or specialized measures particular to the science being performance by require more processing. As part of our BP format, we store these values not only as part of data payload, but also in our index.

1.18.3.2 PG Index table

As shown in Fig. 7, the process group (PG) index table encompasses the count and the total length of all the PGs as the first two entries. The rest of the tables contain a set of information for each PG, which contains the group name information, process ID, and time index. The Process ID specifies which process a group is written by. That process will be the rank value in the communicator if the MPI method is used. Most importantly, there is a file-offset entry for each PG, allowing a fast skip of the file in the unit of the process group.

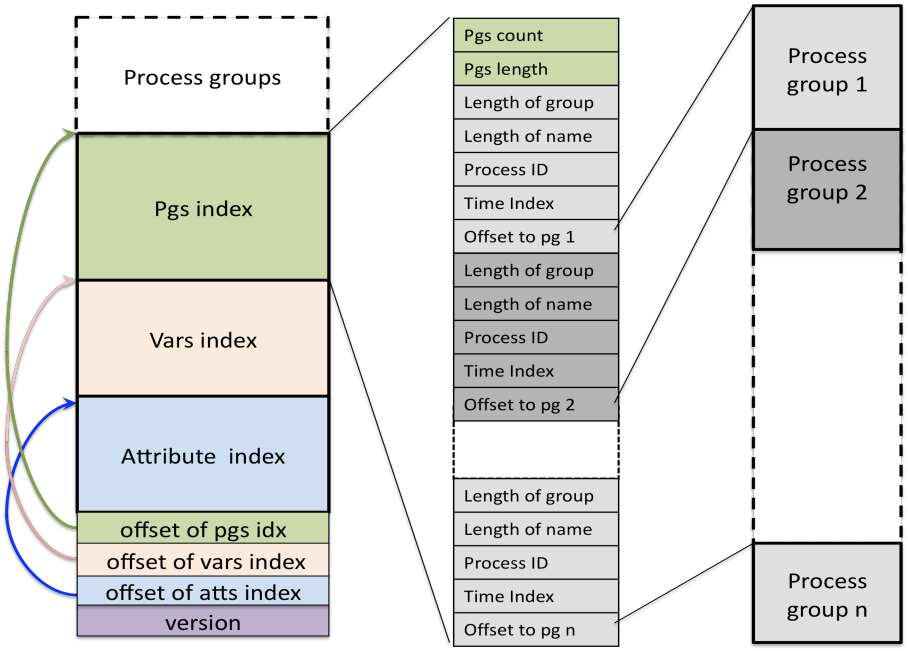

Fig. 7. Group index table.

1.18.3.3 Variables index table

The variables index table is composed of the total count of variables in the BP file, the size of variables index table, and a list of variable records. Each record contains the size of the record and the basic metadata to describe the variable. As shown in Fig. 8, the metadata include the name of the variable, the name of the group the variable is associated with, the data type of the variable, and a series of characteristic features. The structure of each characteristic entry contains an offset value, which is addressed to the certain occurrence of the variable in the BP file. For instance, if n processes write out the variable "d" per time step, and m iterations have been completed during the whole simulation, then the variable will be written (m × n) times in the BP file that is produced. Accordingly, there will be the same number of elements in the list of characteristics. In this way, we can quickly retrieve the single dataset for all time steps or any other selection of time steps. This flexibility and efficiency also apply to a scenario in which a portion of records needs to be collected from a certain group of processes.

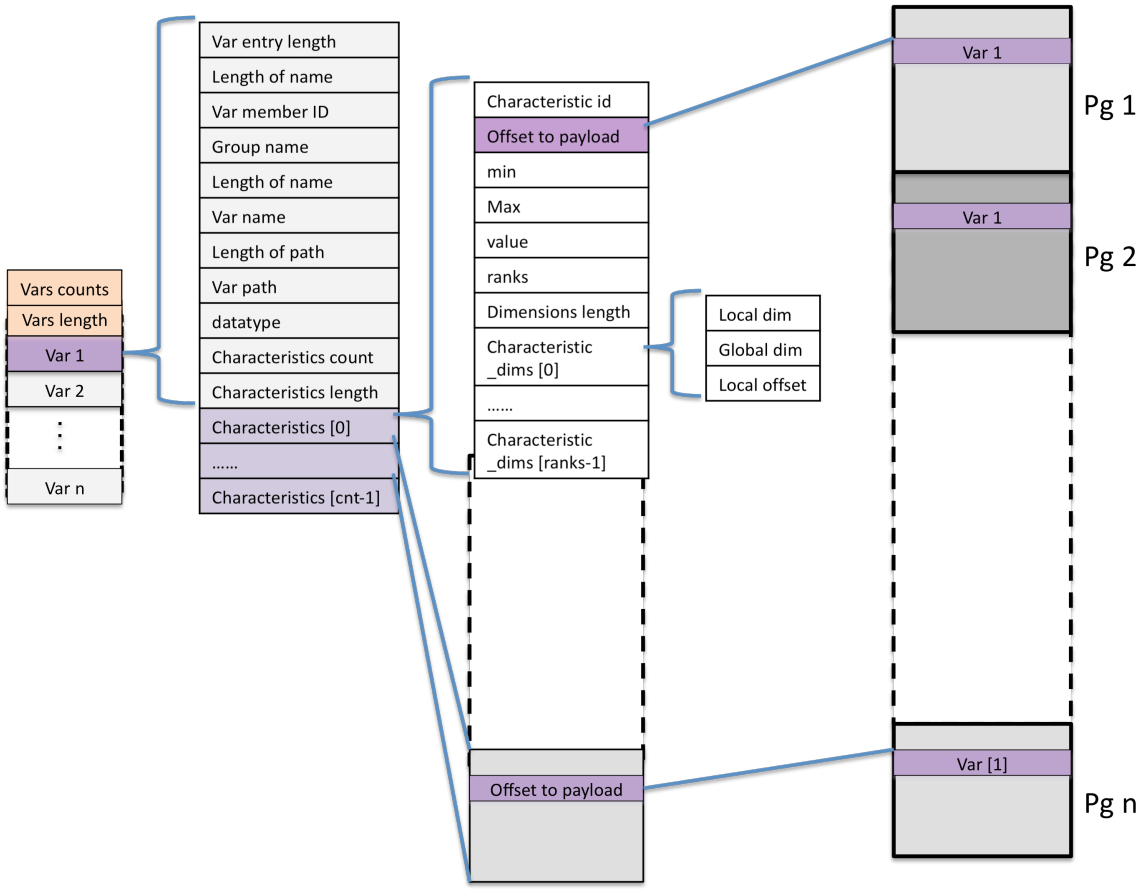

Fig. 8. Vars Index table.

1.18.3.4 Attributes index table

Since an attribute can be considered to be a special type of variable, its index table in BP format is organized in the same way as a variables index table and therefore supports the same types of features mentioned in the previous sections.

1.19 Process Groups

One of the major concepts in BP format is what is called “process group” or PG. The BP file format encompasses a series of PG entries and the BP file footer. Each process group is the entire self-contained output from a single process and is written out independently into a contiguous disk space. In that way, we can enhance parallelism and reduce coordination among processes in the same communication group. The data diagram in Fig. 9 illustrates the detailed content in each PG.

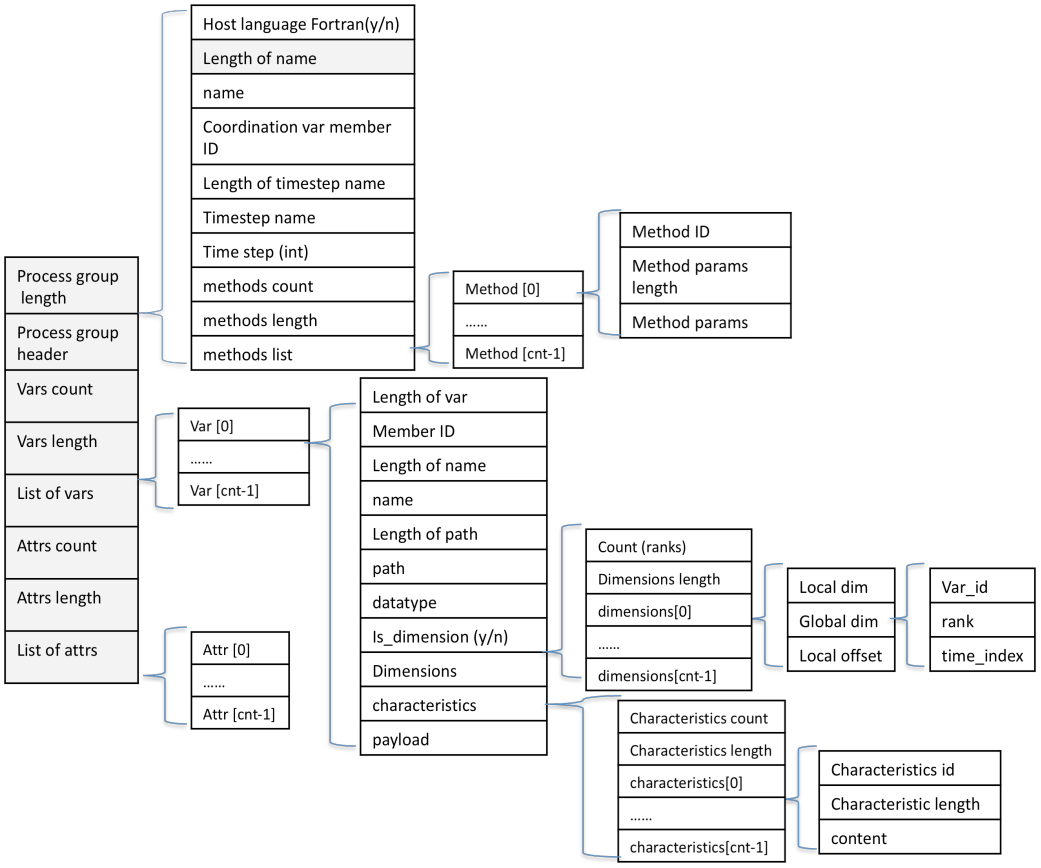

Fig. 9. Process group structure.

1.19.1 PG header

1.19.1.1 Unlimited dimension

BP format allows users to define an unlimited dimension, which will be specified as the time-index in the XML file. Users can define variables having a dimension with undefined length, for which the variable can grow along that dimension. PG is a self-contained, independent data structure; the dataset in the local space per each time step is not reconstructed at the writing operations across the processes or at time steps. Theoretically, PGs can be appended to infinity; they can be added one after another no matter how many processes or time steps take place during the simulation. Thus ADIOS is able to achieve high I/O performance.

1.19.1.2 Transport methods

One of the advantages of organizing output in terms of groups is to categorize all the variables based on their I/O patterns and logical relationships. It provides flexibility for each group to choose the optimized transport method according to the simulation environment and underlying hardware configuration or the transport methods used for a performance study without even changing the source code. In PG header structure, each entry in the method list has a method ID and method parameters, such as system-tuning parameters or underneath driver selection.

1.19.2 Vars list

1.19.2.1 Var header

1.19.2.1.1 Dimensions structure

Internal to bp is sufficient information to recreate any global structure and to place the local data into the structure. In the case of a global array, each process writes the size of the global array dimensions, specifies the local offsets into each, and then writes the local data, noting the size in each dimension. On conversion to another format, such as HDF5, this information is used to create hyperslabs for writing the data into the single, contiguous space. Otherwise, it is just read back in and used to note where the data came from. In this way, we can enhance parallelism and reduce coordination. All of our parallel writes occur independently unless the underlying transport specifically requires collective operations. Even in those cases, the collective calls are only for a full buffer write (assuming the transport was written appropriately) unless there is insufficient buffer space.

As shown in Fig. 9, the dimension structure contains a time index flag, which indicates whether this variable has an unlimited time dimension. Var_id is used to retrieve the dimension value if the dimension is defined as variable in the XML file; otherwise, the rank value is taken as the array dimension.

1.19.2.2 Payload

Basic statistical characteristics give users the advantage for quick data inspection and analysis. In Fig. 9, redundant information about characteristics is stored along with variable payload so that if the characteristics part in the file footer gets corrupted, it can still be recovered quickly. Currently, only simple statistical traits are saved in the file, but the characteristics structure will be easily expanded or modified according to the requirements of scientific applications or the analysis tools.

1.19.3 Attributes list

The layout of the attributes list (see Fig. 10) is very similar to that of the variables. However, instead of containing dimensional structures and physical data load, the attribute header has an is_var flag, which indicates either that the value of the attribute is referenced from a variable by looking up the var_id in the same group or that it is a static value defined in the XML file.

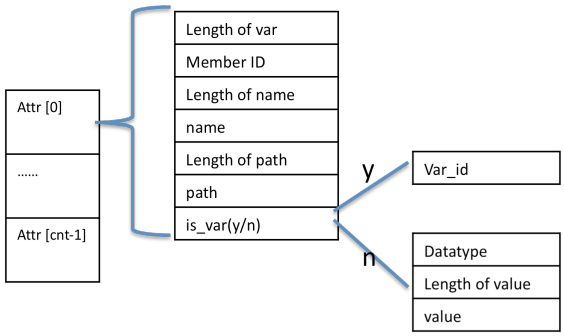

Fig. 10. Attribute entry structure.

Utilities

1.20 adios_lint

We provide a verification tool, called adios_lint, which comes with ADIOS 1.0. It can help users to eliminate unnecessary semantic errors and to verify the integrity of the XML file. Use of adios_lint is very straightforward; enter the adios_lint command followed by the config file name.

1.21 bpdump

The bpdump utility enables users to examine the contents of a bp file and to display all the contents or selected variables in the format on the standard output.

It dumps the bp file content, including the indexes for all the process groups, variables, and attributes, followed by the variables and attributes list of individual process groups (see Fig. 11).

```
bpdump [-d var|--dump var] <filename>

=========================================================
Process Groups Index:
Group: temperature
      Process ID: 0
      Time Name:
      Time: 1
      Offset in File: 0
=========================================================
Vars Index:
Var (Group) [ID]: /NX (temperature) [1]
      Datatype: integer
      Vars Characteristics: 20
Offset(46)          Value(10)
Var (Group) [ID]: /size (temperature) [2]
      Datatype: integer
      Vars Characteristics: 20
      Offset(77)          Value(20)
…
Var (Group) [ID]: /rank (temperature) [3]
      Datatype: integer
      Vars Characteristics: 20
            Offset(110)          Value(0)
…
Var (Group) [ID]: /temperature (temperature) [4]
      Datatype: double
      Vars Characteristics: 20
```

```
        Offset(143)        Min(1.000000e-01)        Max(9.100000e+00)
Dims (l:g:o): (1:20:0,10:10:0)
...
============================================================
Attributes Index:
Attribute (Group) [ID]: /recorded-date (temperature) [5]
    Datatype: string
    Attribute Characteristics: 20
    Offset(363)        Value(Sep-19-2008)
...
```

Fig. 11. bpdump snapshot.

Converters

To make BP files compatible with the popular file formats, we provide a series of converters to convert BP files to HDF5, NETCDF, or ASCII. As long as users give the required schema via the configuration file, the different converter tools currently in ADIOS have the features to translate intermediate BP files to the expected HDF5, NetCDF, or ASCII formats.

1.22 bp2h5

This converter, as indicated by its name, can convert BP files into HDF5 files. Therefore, the same postprocessing tools can be used to analyze or visualize the converted HDF5 files, which have the same data schema as the original ones. The converter can match the row-based or column-based memory layout for datasets inside the file based on which language the source codes are written in. If the XML file specifies global-bounds information, the individual sub-blocks of the dataset from different process groups will be merged into one global the dataset in HDF file.

1.23 bp2ncd

The bp2ncd converter is used to translate bp files into NetCDF files. In Chap. 5, we describe the time-index as an attribute for adios-group. If the variable is time-based, one of its dimensions needs to be specified by this time-index variable, which is defined as an unlimited dimension in the file into which it is to be converted. a NetCDF dimension has a name and a length. If the constant value is declared as a dimension value, the dimension in NetCDF will be named varname_n, in which varname is the name of the variable and n is the nth dimension for that variable. To make the name for the dimension value more meaningful, the users can also declare the dimension value as an attribute whose name can be picked up by the converter and used as the dimension name.

Based on the given global bounds information in a BP file, the converter can also reconstruct the individual pieces from each process group and create the global space array in NetCDF. A final word about editing the XML file: the name string can contain only letters, numbers or underscores ("_"). Therefore, the attribute or variable name should conform to this rule.

1.24 bp2ascii

Sometimes, scientists want one variable with all the time steps or want to extract two variables at the same time steps to and store the resulting data in ASCII format. The Bp2ascii converter tool allows users to accomplish those tasks.

Bp2ascii bp_filename –v x1 … xn [–c/-r] –t m,n

-v – specify the variables need to be printed out in ASCII file

-c –print variable values for all the time steps in column

-r – print variable values for all the time steps in row

-t – print variable values for time step m to n, if not defined, all the time steps will be printed out.

1.25 Parallel Converter Tools

Currently, all of the converters mentioned above can only sequentially parse bp files. We will work on developing parallel versions of all of the converters for improved performance of ADIOS 2.0 . As a result, the extra conversion cost to translate bp into the expected file format can be unnoticeable compared with the file transfer time.

Group read/write process

In ADIOS 1.0, we provide a python script, which takes a configuration file name as an input argument and produces a series of preprocessing files corresponding to the individual adios-group in the XML file. Depending on which language (C or FORTRAN) is specified in XML, the python script either generates files gwrite_groupname.ch and gread_groupname.ch for C or files with extension .fh for Fortran. These files contain the size calculation for the group and automatically print adios_write calls for all the variables defined inside adios-group. Using only one "#include filename.ch/filename.fh" statement in source code between the pair of adios_open and adios_close.

Users either type the following command line or incorporate it into Makefie:

```
python gpp.py <config_fname>
```

1.26 Gwrite/gread/read

Below are a few example of the mapping from var element to adios_write/read:

In adios-group "weather", we have a variable declared in the following forms:

1) <var name="temperature" gwrite="t" gread="t_read" type="adios_double" dimensions="NX"/>

When the python command is executed, two files are produced, gwrite_weather.ch and gread_weather.ch. The gwrite_weather.ch command contains

```
adios_write (adios_handle, "temperature", t);
```

while gread_weather.ch contains

```
adios_read (adios_handle, "temperature", t_read).
```

2) <var name="temperature" gwrite="t" gread="t_read" type="adios_double" dimensions="NX" read="no"/>

In this case, only the adios_write statement is generated in gwrite_weather.ch. The adios_read statement is not generated because the value of attribute read is set to no.

3) <var name="temperature" gread="t_read" type="adios_double" dimensions="NX" />

```
adios_write (adios_handle, "temperature", temperature)

adios_read (adios_handle, "temperature", t_read).
```

4) <var name="temperature" gwrite="t" type="adios_double" dimensions="NX" />

```
adios_write (adios_handle, "temperature", t)

adios_read (adios_handle, "temperature", temperature)
```

1.27 Add conditional expression

Sometimes, the adios_write routines are not perfectly written out one after another. There might be some conditional expressions or loop statements. The following example will show you how to address this type of issue via XML editing.

<gwrite src="if (rank == 0) {"/>

<var name="temperature" gwrite="t" gread="t_read" type="adios_double" dimensions="NX" read="no"/>

<gwrite src="}"/>

Rerun the python command; the following statements will be generated in gwrite_weather.ch,

```
if (mype==0) {

adios_write (adios_handle, "temperature", t)

}
```

gread_weather.ch has same condition expression added.

1.28 Dependency in Makefile

Since we include the header files in the source, the users need to include the header files as a part of dependency rules in the Makefile.

C Programming with ADIOS

This chapter focuses on how to integrate ADIOS into the users' source code in C and how to write into separate files or a shared file from multiple processes in the same communication domain.

1.29 Non-ADIOS Program

The programming example shown in Fig. 12 illustrates how to write a double-precision array `t` and a double-precision array with size of `NX` into a file called "test.bp," which is organized in our native tagged binary file format—BP (see Chap. 5). This file format allows users to include rich metadata associated with a block of binary data as well the indexing mechanism for different blocks of data. (See Fig. 3 for the corresponding XML configuration file required by the program.)

```
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank;
    int         NX = 10;
    double      t[NX];
    FILE        * fp;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    sprintf (filename, "restart_%5.5d.bp", rank);
    fp = open (filename, "w");
    fwrite ( &NX, sizeof(int), 1, fp);
    fwrite (t,  sizeof(double), NX, fp);
    fclose (fp);

    MPI_Finalize ();
    return 0;
}
```

Fig. 12. Example of a user's original program.

1.30 Construct an XML File

From the example routine, we know that the program is designed to write a file for each process. There is a double-precision one-dimensional array called "t". Therefore, our configuration file is constructed as shown in Fig. 13.

```
/* config.xml*/
<adios-config host-language="C">
  <adios-group name="temperature" coordination-communicator="comm">
  <var name="NX" type="integer"/>
  <var name="temperature" gwrite="t" type="double" dimensions="NX"/>
  <attribute name="recorded-date" path="/" value="Sep-19-2008"
type="string"/>
</adios-group>

<method group="temperature" method="MPI"/>

<buffer size-MB="1" allocate-time="now"/>

</adios-config>
```

Fig. 13. Example config.xml file.

1.31 Generate .ch file (s)

The adios_group_size function and a set of adios_write functions can be automatically generated in gwrite_temperature.ch file by using the following python command (see Chap. 10):

```
python gpp.py config.xml
```

The generated gwrite_temperature.ch file is given in Fig. 14.

```
/* gwrite_temperature.ch */
adios_groupsize = 4 \
          + 8 * (NX);
adios_group_size (adios_handle, adios_groupsize, &adios_totalsize, &comm);
adios_write (adios_handle, "NX", &NX);
adios_write (adios_handle, "temperature", t);
```

Fig. 14. Example gwrite_temperature.ch file.

1.32 Write to Separate Files for each Process (P writers, P files)

For our first program, we will simply translate the program of Fig 14 so that all of the I/O operations are done with ADIOS routines. We do this to show how familiar and easy I/O operations look in ADIOS. This program has the same advantages and disadvantages as the preceding version. Consider the differences between programs shown in Figs. 15 and 16 one by one; there are only four.

1.32.1 POSIX

We show how to use the POSIX method to write out separate files for each processor in Fig. 15.

```
/*write Separate file for each process by using POSIX*/
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank;
    int         NX = 10;
    double      t[NX];

    /* ADIOS variables declarations for matching gwrite_temperature.ch */
    int             adios_err;
    uint64_t        adios_groupsize, adios_totalsize;
    int64_t         adios_handle;
    MPI_Comm  * comm =  MPI_COMM_SELF;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    sprintf (filename, "restart_%5.5d.bp", rank);
    adios_init ("config.xml");
    adios_open (&adios_handle, "temperature", filename, "w");
    #include "gwrite_temperature.ch"
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Fig. 15. Example adios P2P program.

1.32.2 MPI-IO

Based on the same group description in the configure file and the header file (.ch) generated by python script, we can switch among different transport methods without changing or recompiling the source code.

One entry change in the XML file can switch from POSIX to MPI-IO when the source code is recompiled:

<method group="temperature" method="**MPI**"/>

MPI communicator is passed as an argument of adios_group_size call in "gwrite_temperature.ch". Because it is defined as MPI_COMM_SELF, every process writes out its own file.

There are several ways to verify the binary results. We can either choose bpdump to display the content of the file or use one of the converters (bp2ncd, bp2h5, or bp2ascii), to produce the user's preferred file format (NetCDF, HDF5 or ASCII, respectively) and use its dump utility to output the content in the standard output.

1.33 Writing to Shared Files (P writers, N files)

As the number of processes increases to 10,000, the amount of files will increase by the same magnitude if we use the basis P to P method. All the files will be difficult to manage; the independent POSIX I/O operations will probably give the best performance. Now we will address a scenario in which multiple processes write to N files. In the following example (Fig. 16), we write out only two files with P processes. All the even-ranked processes write to a shared file while the odd-ranked processes write to another file.

```
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank, size;
    int         NX = 10;
    double      t[NX];

    /* ADIOS variables declarations for matching gwrite_temperature.ch */
    int             adios_err;
    uint64_t        adios_groupsize, adios_totalsize;
    int64_t         adios_handle;
    MPI_Comm comm;
    /*
    int         color, key;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    /* MPI_Comm_split partitions the world group into two disjointed  subgroups,
     * the processes are ranked in terms of the argument key
     *  a new communicator comm is returned for this specific grid configuration
     */
    color = rank % 2;
```

```
    key = rank / 2;
    MPI_Comm_split (MPI_COMM_WORLD, color, key, &comm);

    /* every P/2 processes write into the same file
     * there are 2 files generated.
     */
    sprintf (filename, "restart_%5.5d.bp", color);
    adios_init ("config.xml");
    adios_open (&adios_handle, "temperature", filename, "w");
    #include "gwrite_temperature.ch"
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Fig. 16. Example ADIOS P2N program.

The reconstructed MPI communicator `comm` is passed as an argument of the adios_group_size call in "gwrite_temperature.ch". Therefore, in this example, each file is written by the processes in the same communication domain.

There is no need to change the XML file in this case because we are still using the MPI method.

1.34 Writing to Shared Files with Collective I/O

If users prefer to use the MPI collective I/O rather than an independent I/O, all that is necessary is to change the value of the attribute "method" to MPI-CIO in XML, and the results will be generated by the MPI-CIO method.

```
<method group="temperature" method="MPI-CIO"/>
```

1.35 Global Arrays

If each process writes out a subarray that belongs to the same global space, ADIOS provides the way to write out global information and to reconstruct it into a whole global array in HDF5 or NetCDF file when using our converters. Figures 17–19 show how to write global arrays. Figure 17 is an example of how to edit an **XML file.**

```
/* config.xml*/
<adios-config host-language="C">
  <adios-group name="temperature" coordination-communicator="comm">
  <var name="NX" type="integer"/>
  <var name="size" type="integer"/>
  <var name="key" type="integer"/>
```

```
  <global-bounds dimensions="size,NX" offsets="key,0">
    <var name="temperature" gwrite="t" type="double" dimensions="1,NX"/>
  </global-bounds>
  <attribute name="recorded-date" path="/" value="Sep-19-2008"
type="string"/>
</adios-group>
<method group="temperature" method="MPI"/>
<buffer size-MB="1" allocate-time="now"/>
</adios-config>
```

Fig. 17. Example of how to edit an XML file.

Because the XML configuration has been modified, we need to rerun the python command to generate the corresponding header file (see Fig. 18).

```
/* gwrite_temperature.ch */
adios_groupsize = 4 \
                  + 4 \
                  + 4 \
                  + 8 * (1) * (NX);
adios_group_size (adios_handle, adios_groupsize, &adios_totalsize, &comm);
adios_write (adios_handle, "NX", &NX);
adios_write (adios_handle, "size", &size);
adios_write (adios_handle, "rank", &rank);
adios_write (adios_handle, "temperature", t);
```

Fig. 18. Example of how to edit a python script to generate the header script.

Having edited the XML file and the python script, we can run the code and convert the output to verify the results. For instance, we can use the program demonstrated in Sect. 10.7 to generate new bp files. To verify the resulting bp file with correct global information, the bp2ncd is used to convert the bp file to an NetCDF file. Figure 19 is the result for restart.nc.

```
netcdf restart { // format variant: 64bit
dimensions:
    NX = 10 ;
    size = 20 ;
    rank = 1 ;
variables:
    double temperature(size, NX) ;
```

```
// global attributes:
            :recorded-date = "Sep-19-2008" ;
data:

 temperature =
  0.1, 1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1,
  …
  0.1, 20.1, 40.1, 60.1, 80.1, 100.1, 120.1, 140.1, 160.1, 180.1 ;
}
```

Fig. 19. Converted results file.

1.36 Writing Time-Index into a Variable

The time-index allows user to define a variable having a dimension with an undefined length, along which the variable can grow. Say users want to write out temperature after a certain number of iterations (Fig. 20). First, we add the "time-index" attribute in adios-group called "time." Next, we find the variable temperature in the adios-group and add "time" as an extra dimension for it; the record number for that variable will be stored every time it gets written out.

```
/* config.xml*/
<adios-config host-language="C">
   <adios-group name="temperature" coordination-communicator="comm" time-
index="time">
   <var name="NX" type="integer"/>
   <var name="size" type="integer"/>
   <var name="key" type="integer"/>
   <global-bounds dimensions="size,NX" offsets="key,0">
     <var name="temperature" gwrite="t" type="double" dimensions="1,NX,
time"/>
   </global-bounds>
   <attribute name="recorded-date" path="/" value="Sep-19-2008"
type="string"/>
</adios-group>
<method group="temperature" method="MPI"/>
<buffer size-MB="1" allocate-time="now"/>
</adios-config>
```

Fig. 20. Example of a file with a time variable added.

The advantage of ADIOS is that the user does not need change and recompile the code; the variable address and the size of the variable have not changed. The user submits the job and generates the new bp files.

To verify the results, the users can use bpdump to examine the bp file content or use the bp2ncd converter to view the content as an NetCDF File.

1.37 Reading the File

Now let's move to examples of how to read the data from BP or other files. Assuming that we still use the same configure file shown in Fig. 13, the following steps illustrate how to easily change the code and xml file to read a variable.

1. add another variable adios_buf_size specifying the size for read.
2. call adios_open with "r" (read only) mode.
3. Insert #include "gread_temperature.ch"
4. Rerun the gpp.py and generate the file gread_temperature.ch

Figure 21 shows C routines integrated with ADIOS APIs.

```
/*write Separate file for each process by using POSIX*/
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank;
    int         NX = 10;
    double      t[NX];

    /* ADIOS variables declarations for matching gread_temperature.ch */
    int             adios_err;
    uint64_t        adios_groupsize, adios_totalsize, adios_buf_size;
    int64_t         adios_handle;
    MPI_Comm  * comm =  MPI_COMM_SELF;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    sprintf (filename, "restart_%5.5d.bp", rank);
    adios_init ("config.xml");
    adios_open (&adios_handle, "temperature", filename, "r");
    #include "gread_temperature.ch"
```

```
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Fig. 21. Example of C routines integrated with ADIOS APIs for gread_temperature.ch.

The generated gread_temperature.ch file is given in Fig. 22.

```
/* gread_temperature.ch */
adios_group_size (adios_handle, adios_groupsize, &adios_totalsize, &comm);
adios_buf_size = 4;
adios_read (adios_handle, "NX", &NX, adios_buf_size);
adios_buf_size = NX;
adios_read (adios_handle, "temperature", t, adios_buf_size);
```

Fig. 22. Example of a generated gread_temperature.ch file.

Advanced Programming with ADIOS

If the users are more interested in advanced programming and would like to contribute their efforts to the development or customization of the ADIOS library, this section will touch on some advanced concepts in ADIOS.

1.38 Asynchronous I/O Programming Model

Figure 23 shows the basic programming model of how to implement non-blocking IO.

```
adios_init ("config.xml")
...
// do main loop

adios_begin_calculation ()
// do non-communication work
adios_end_calculation ()

// perform restart writ
// do communication work
adios_end_iteration ()

! end loop
...
adios_finalize (myproc_id)
```

Fig. 23. Example of asynchronous programming.

1.39 Create New Transport Methods

One of ADIOS's important features is the componentization of transport methods. Users can switch among the typical methods that we support or even create their own methods, which can be easily plugged into our library. The following sections provide the procedures for adding the new transport method called "abc" into the ADIOS library. In this version of ADIOS, all the source files are located in /trunk/src/.

1.39.1 Add the new method macros in adios_transport_hooks.h

The first file users need to examine is adios_transport_hooks.h, which basically defines all the transport methods and interface functions between detailed transport implementation and user APIs. In the file, we first find the line that defines the enumeration type Adios_IO_methods_datatype add the declaration of method ID ADIOS_METHOD_ABC, and, because we add a new method, update total number of transport methods ADIOS_METHOD_COUNT from 9 to 10.

1. enum Adios_IO_methods datatype

```
enum ADIOS_IO_METHOD {
        ADIOS_METHOD_UNKNOWN  = -2
       ,ADIOS_METHOD_NULL     = -1
       ,ADIOS_METHOD_MPI      = 0
       ……
       ,ADIOS_METHOD_PHDF5    = 8
                                  ← ADIOS_METHOD_ABC  = 9
       ,ADIOS_METHOD_COUNT    = 9  ← ADIOS_METHOD_COUNT  = 10
       };
```

2. Next, we need to declare the transport APIs for method "abc," including init/finalize, open/close, should_buffer, and read/write. Similar to the other methods, we need to add

```
FORWARD_DECLARE (abc)
```

3. Then, we add the mapping of the string name "abc" of the new transport method to the method ID - ADIOS_METHOD_ABC, which has been already defined in enumeration type Adios_IO_methods_datatype. As the last parameter, "1" here means the method requires communications, or "0" if not.

```
MATCH_STRING_TO_METHOD ("abc", ADIOS_METHOD_ABC, 1)
```

4. Lastly, we add the mapping of the string name needed in the initialization functions to the method ID, which will be used by adios_transport_struct variables defined in adios_internals.h.

ASSIGN_FNS (abc, ADIOS_METHOD_ABC)

1.39.2 Create adios_abc.c

In this section, we demonstrate how to implement different transport APIs for method "abc." In adios_abc.c, we need to implement at least 11 required routines:

1. "adios_abc_init" allocates the method_data field in adios_method_struct to the user-defined transport data structure, such as adios_abc_data_struct, and initializes this data structure. Before the function returns, the initialization status can be set by statement "adios_abc_initialized = 1."

2. "adios_abc_open" opens a file if there is only one processor writing to the file. Otherwise, this function does nothing; instead, we use adios_abc_should_buffer to coordinate the file open operations.

3. "adios_abc_should_buffer," called by the "common_adios_group_size" function in adios.c, needs to include coordination of open operations if multiple processes are writing to the same file by using the communicator variable passed as function parameter.

4. "adios_abc_write", in the case of no buffering or overflow, writes data directly to disk. Otherwise, it verifies whether the internally recorded memory pointer is consistent with the vector variable's address passed in the function parameter and frees that block of memory if it is not needed any more.

5. "adios_abc_read" associates the internal data structure's address to the variable specified in the function parameter.

6. "adios_abc_close" closes the file if no buffering scheme is used. Otherwise, this function needs extra effort to perform the actual disk writing/reading to/from the file by one or more processors in the same communicator domain and then close the file.

7. "adios_abc_finalize" resets the initialization status back to 0 if it has been set to 1 by adios_abc_init.

If you are developing asynchronous methods, the following functions need to be implemented as well; otherwise you can leave them as empty implementation.

8. adios_abc_get_write_buffer,

9. "adios_abc_end_iteration" is a tick counter for the I/O routines to time how fast they are emptying the buffers.

10. “adios_abc_start_calculation” indicates that it is now an ideal time to do bulk data transfers because the code will not be performing I/O for a while.

11. “adios_abc_stop_calculation“ indicates that bulk data transfers should cease because the code is about to start communicating with other nodes.

The following is One of the most important things that needs to be noted:

fd->shared_buffer = adios_flag_no,

which means that the methods do not need a buffering scheme, such as PHDF5, and that data write out occurs immediately once adios_write returns.

If fd->shared_buffer = adios_flag_yes, the users can employ the self-defined buffering scheme to improve I/O performance.

1.39.3 A walk-through example

Now let’s look at an example of adding an unbuffered POSIX method to ADIOS. According to the steps described above, we first open the header file -- “adios_transport_hooks.h,” and add the following statements:

- **enum ADIOS_IO_METHOD** {

  ADIOS_METHOD_UNKNOWN = -2
  ,ADIOS_METHOD_NULL = -1
  ,ADIOS_METHOD_MPI = 0
  …
  ,ADIOS_METHOD_PROVENANCE = 8
  // method ID for binary transport method
  ,ADIOS_METHOD_POSIX_ASCII_NB = 9
  // total method number
  ,ADIOS_METHOD_COUNT = 10
  };

- **FORWARD_DECLARE (posix_ascii_nb);**

- **MATCH_STRING_TO_METHOD ("posix_ascii_nb"**

  , ADIOS_METHOD_ POSIX_ASCII_NB, 0)

- **ASSIGN_FNS (binary, ADIOS_METHOD_ POSIX_ASCII_NB)**

Next, we must create adios_posix_ascii_nb,c, which defines all the required routines listed in Sect. 12.1.2 The blue highlights below mark out the data structures and required functions that developers need to implement in the source code.

---

static int **adios_posix_ascii_nb_initialized** = 0;

```
struct adios_POSIX_ASCII_UNBUFFERED_data_struct
{
  FILE *f;
  uint64_t file_size;
};

void adios_ posix_ascii_nb _init (const char *parameters
                                  , struct adios_method_struct * method)
{
  struct adios_POSIX_ASCII_UNBUFFERED_data_struct * md;
  if (!adios_posix_ascii_nb_initialized)
  {
    adios_posix_ascii_nb_initialized = 1;
  }
  method->method_data = malloc (
                          sizeof(struct adios_POSIX_ASCII_UNBUFFERED_data_struct)
                                );
  md = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                            method->method_data;
  md->f = 0;
  md->file_size = 0;
}


int adios_ posix_ascii_nb _open (struct adios_file_struct * fd
                        , struct adios_method_struct * method)
{
  char * name;
  struct adios_POSIX_ASCII_UNBUFFERED_data_struct * p;
  struct stat s;
  p = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                                    method->method_data;
  name = malloc (strlen (method->base_path) + strlen (fd->name) + 1);
  sprintf (name, "%s%s", method->base_path, fd->name);
  if (stat (name, &s) == 0)
    p->file_size = s.st_size;
  switch (fd->mode)
  {
    case adios_mode_read:
    {
      p->f = fopen (name, "r");
      if (p->f <= 0)
      {
        fprintf (stderr, "ADIOS POSIX ASCII UNBUFFERED: "
                 "file not found: %s\n", fd->name);
        free (name);
```

```
            return 0;
          }
          break;
        }
        case adios_mode_write:
        {
          p->f = fopen (name, "w");
          if (p->f <= 0)
          {
            fprintf (stderr, "adios_posix_ascii_nb_open "
                     "failed for base_path %s, name %s\n"
                    ,method->base_path, fd->name
                    );
            free (name);
            return 0;
          }
          break;
        }
        case adios_mode_append:
        {
          int old_file = 1;
          p->f = fopen (name, "a");
          if (p->f <= 0)
          {
            fprintf (stderr, "adios_posix_ascii_nb_open"
                     " failed for base_path %s, name %s\n"
                    ,method->base_path, fd->name
                    );
            free (name);
            return 0;
          }
          break;
        }
        default:
        {
          fprintf (stderr, "Unknown file mode: %d\n", fd->mode);
          free (name);
          return 0;
        }
      }
      free (name);
      return 0;
    }
    enum ADIOS_FLAG adios_ posix_ascii_nb _should_buffer
                                    (struct adios_file_struct * fd
                                    ,struct adios_method_struct * method
```

```
                                    ,void * comm)
{
  //in this case, we don't use shared_buffer
  return adios_flag_no;
}
void adios_abc_write (struct adios_file_struct * fd
                      ,struct adios_var_struct * v
                      ,void * data
                      ,struct adios_method_struct * method )
{
  struct adios_POSIX_ASCII_UNBUFFERED_data_struct * p;
  p = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                        method->method_data;
  if (!v->dimensions) {
    switch (v->type)
    {
      case adios_byte:
      case adios_unsigned_byte:
        fprintf (p->f,"%c\n", *((char *)data));
        break;
      case adios_short:
      case adios_integer:
      case adios_unsigned_short:
      case adios_unsigned_integer:
        fprintf (p->f,"%d\n", *((int *)data));
        break;
      case adios_real:
      case adios_double:
      case adios_long_double:
        fprintf (p->f,"%f\n", *((double *)data));
        break;
      case adios_string:
        fprintf (p->f,"%s\n", (char *)data);
        break;
      case adios_complex:
        fprintf (p->f,"%f+%fi\n", *((float *)data),*((float *)(data+4)));
        break;
      case adios_double_complex:
        fprintf (p->f,"%f+%fi\n", *((double *)data),*((double *)(data+8)));
        break;
         default:
        break;
     }
  }
  else
  {
```

```
uint64_t j;
int element_size = adios_get_type_size (v->type, v->data);
printf("element_size: %d\n",element_size);
uint64_t var_size = adios_get_var_size (v, fd->group, v->data)/element_size;
switch (v->type)
{
   case adios_byte:
   case adios_unsigned_byte:
      for (j = 0;j < var_size; j++)
         fprintf (p->f,"%c ", *((char *)(data+j)));
      printf("\n");
      break;
   case adios_short:
   case adios_integer:
   case adios_unsigned_short:
   case adios_unsigned_integer:
      for (j = 0;j < var_size; j++)
         fprintf (p->f,"%d ", *((int *)(data+element_size*j)));
      printf("\n");
      break;
   case adios_real:
   case adios_double:
   case adios_long_double:
      for (j = 0;j < var_size; j++)
         fprintf (p->f,"%f ", * ( (double *)(data+element_size*j)) );
      printf("\n");
      break;
   case adios_string:
      for (j = 0;j < var_size; j++)
         fprintf (p->f,"%s ", (char *)data);
      printf("\n");
      break;
   case adios_complex:
      for (j = 0;j < var_size; j++)
         fprintf (p->f, "%f+%fi ", *((float *)(data+element_size*j))
               ,*((float *)(data+4+element_size*j))
               );
      printf("\n");
      break;
   case adios_double_complex:
      for (j = 0;j < var_size; j++)
         fprintf (p->f,"%f+%fi ", *((double *)(data+element_size*j))
               ,*((double *)(data+element_size*j+8)));
      printf("\n");
      break;
   default:
```

```
            break;
        }
    }
}

void adios_posix_ascii_nb _get_write_buffer
                                    (struct adios_file_struct * fd
                                     ,struct adios_var_struct * v
                                     ,uint64_t * size
                                     ,void ** buffer
                                     ,struct adios_method_struct * method)
{
   *buffer = 0;
}

void adios_ posix_ascii_nb _read (struct adios_file_struct * fd
                        ,struct adios_var_struct * v, void * buffer
                         ,uint64_t buffer_size
                         ,struct adios_method_struct * method )
{
   v->data = buffer;

   v->data_size = buffer_size; }

int adios_posix_ascii_nb _close (struct adios_file_struct * fd
                                    , struct adios_method_struct * method)
{
struct adios_POSIX_ASCII_UNBUFFERED_data_struct * p;
   p = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                                        method->method_data;
   if (p->f <= 0)
   {
      fclose (p->f);
   }
   p->f = 0;
   p->file_size = 0;
}

void adios_posix_finalize (int mype, struct adios_method_struct * method)
{
   if (adios_posix_ascii_nb_initialized)
      adios_posix_ascii_nb_initialized = 0;
}
```

The binary transport method blocks methods for simplicity. Therefore, no special implementation for the three functions below is necessary and their function bodies can be left empty:

adios_abc_end_iteration (struct adios_method_struct * method) {}
adios_abc_start_calculation (struct adios_method_struct * method) {}
adios_abc_stop_calculation (struct adios_method_struct * method) {}

Until now, we have implemented the POSIX_ASCII transport method. When users specify POSIX_ASCII_NB method in xml file, the users' applications will generate ASCII files by using common ADIOS APIs. However, in order to achieve better I/O performance, a buffering scheme needs to be included into this example.

1.40 Profiling the Application and ADIOS

There are two ways to get profiling information of ADIOS I/O operations. One way is for the user to explicitly insert a set of profiling API calls around ADIOS API calls in the source code. The other way is to link the user code with a renamed ADIOS library and an ADIOS API wrapper library.

1.40.1 Use profiling API in source code

The profiling library called libadios_timing.a implements a set of profiling API calls. The user can use these API calls to wrap the ADIOS API calls in the source code to get profiling information.

The adios-timing.h header file contains the declarations of those profiling functions.

```
/*
 * initialize profiling
 *
 * Fortran interface
 */
int init_prof_all_(char *prof_file_name, int prof_file_name_size);

/*
 * record open start time for specified group
 *
 * Fortran interface
 */
void open_start_for_group_(int64_t *gp_prof_handle, char *group_name, int
*cycle, int *gp_name_size);

/*
 * record open end time for specified group
 *
 * Fortran interface
```

```
 */
void open_end_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record write start time for specified group
 *
 * Fortran interface
 */
void write_start_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record write end time for specified group
 *
 * Fortran interface
 */
void write_end_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record close start time for specified group
 *
 * Fortran interface
 */
void close_start_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record close end time for specified group
 *
 * Fortran interface
 */
void close_end_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * Report timing info for all groups
 *
 * Fortran interface
 */
int finalize_prof_all_();

/*
 * record start time of a simulation cycle
 *
 * Fortran interface
 */
void cycle_start_(int *cycle);
```

```
/*
 * record end time of a simulation cycle
 *
 * Fortran interface
 */
void cycle_end_(int *cycle);
```

An example of using these functions is given below.

```
…
! initialization ADIOS
CALL adios_init ("config.xml"//char(0))
! initialize profiling library; the parameter specifies the file where profiling
information is written
CALL init_prof_all("log"//char(0))
…
CALL MPI_Barrier(toroidal_comm, error )

! record start time of open
! group_prof_handle is an OUT parameter holding the handle for the group
‘output3d.0’
! istep is iteration no.
CALL open_start_for_group(group_prof_handle, "output3d.0"//char(0),istep)

CALL adios_open(adios_handle, "output3d.0"//char(0), “w”//char(0))

! record end time of open
CALL open_end_for_group(group_prof_handle,istep)

! record start time of write
CALL write_start_for_group(group_prof_handle,istep)

#include "gwrite_output3d.0.fh"

! record end time of write
CALL write_end_for_group(group_prof_handle,istep)

! record start time of close
CALL cose_start_for_group(group_prof_handle,istep)

CALL adios_close(adios_handle,adios_err)

! record end time of close
CALL close_end_for_group(group_prof_handle,istep)
```

```
…
CALL adios_finalize (myid)

! finalize; profiling information are gathered and min/max/mean/var are
calculated for each IO dump
CALL finalize_prof()

CALL MPI_FINALIZE(error)
```

When the code is run, profiling information will be saved to the file "./log" (specified in init_prof_all ()). Below is an example.

```
Fri Aug 22 15:42:04 EDT 2008
I/O Timing results
Operations  :      min              max              mean             var
cycle no      3
io count      0
# Open      :      0.107671         0.108245         0.108032         0.000124
# Open start :     1219434228.866144    1219434230.775268    1219434229.748614    0.588501
# Open end   :     1219434228.974225    1219434230.883335    1219434229.856646    0.588486
# Write     :      0.000170         0.000190         0.000179         0.000005
# Write start :    1219434228.974226    1219434230.883336    1219434229.856647    0.588486
# Write end  :     1219434228.974405    1219434230.883514    1219434229.856826    0.588484
# Close     :      0.001608         0.001743         0.001656         0.000036
# Close start :    1219434228.974405    1219434230.883514    1219434229.856826    0.588484
# Close end  :     1219434228.976040    1219434230.885211    1219434229.858482    0.588489
# Total     :      0.109484         0.110049         0.109868         0.000137
cycle no      6
io count      1
# Open      :      0.000007         0.000011         0.000009         0.000001
# Open start :     1219434240.098444    1219434242.007951    1219434240.981075    0.588556
# Open end   :     1219434240.098452    1219434242.007962    1219434240.981083    0.588556
# Write     :      0.000175         0.000196         0.000180         0.000004
# Write start :    1219434240.098452    1219434242.007962    1219434240.981083    0.588557
# Write end  :     1219434240.098631    1219434242.008158    1219434240.981264    0.588558
# Close     :      0.000947         0.003603         0.001234         0.000466
# Close start :    1219434240.098631    1219434242.008158    1219434240.981264    0.588558
# Close end  :     1219434240.099665    1219434242.009620    1219434240.982498    0.588447
# Total     :      0.001132         0.003789         0.001423         0.000466
```

The script "post_script.sh" extracts open time, write time, close time, and total time from the raw profiling results and saves them in separate files: open, write, close, and total, respectively.

To compile the code, one should link the code with the *–ladios_timing -ladios* option.

1.40.2 Use wrapper library

Another way to do profiling is to link the source code with a renamed ADIOS library and a wrapper library.

The renamed ADIOS library implements "real" ADIOS routines, but all ADIOS public functions are renamed with a prefix "P". For example, adios_open() is

renamed as Padios_open(). The routine for parsing config.xml file is also changed to parse extra flags in config.xml file to turn profiling on or off.

The wrapper library implements all adios pubic functions (e.g., adios_open, adios_write, adios_close) within each function. It calls the "real" function (Padios_xxx()) and measure the start and end time of the function call.

There is an example wrapper library called libadios_profiling.a. Developers can implement their own wrapper library to customize the profiling.

To use the wrapper library, the user code should be linked with *–ladios_profiling –ladios*. the wrapper library should precede the "real" ADIOS library. There is no need to put additional profiling API calls in the source code. The user can turn profiling on or off for each ADIOS group by setting a flag in the config.xml file.

```
<adios-group name="restart.model" profiling="yes|no">
   ...
</adios-group>
```

FAQs

1.41 XML Editing

1.42 Programming

1.43 Debugging

1.44 Method Switching

References

Appendix

Datatypes used in the ADIOS XML file

size	Signed type	Unsigned type
1	byte, interger*1	unsigned byte, unsigned integer*1
2	short, integer*2	unsigned short, unsigned integer*2
4	integer, integer*4, real, real*4, float	unsigned integer, unsigned integer*4
8	long, integer*8, real*8, double, long float, complex, complex*8	
16	real*16, long double, double complex, complex*16	
	string	

ADIOS APIs List

Function	Purpose
adios_init	Load the XML configuration file creating internal representations of the various data types and defining the methods used for writing.
adios_finalize	Cleanup anything remaining before exiting the code
adios_open	Prepare a data type for subsequent calls to write data using the io_handle. Mode is one of “r” (read), “w” (write) and “a” (append).
adios_close	Commit all the write to disk, close the file and release adios file handle
Adios_group_size	Passing the required buffer size and communication coordinator to the transport layer and returned the total size back to the source code
Adios_write	Submit a data element for writing. This does NOT actually perform the write in buffered mode. In the overflow case, this call writes to buffer directly.
Adios_read	Submit a buffer space (var) for reading a data element into. This does NOT actually perform the read. Actual population of the buffer space will happen on the call to

	adios_close
Adios_set_path	Set the HDF5-style path for all variables in a adios-group. This will reset whatever is specified in the XML file.
Adios_set_path_var	Set the HDF-5-style path for the specified var in the group. This will reset whatever is specified in the XML file.
Adios_get_write_buffer	For the given field, get a buffer that will be used at the transport level for it of the given size. If size == 0, then auto calculate the size based on what is known from the datatype in the XML file and any provided additional elements (such as array dimension elements). To return this buffer, just do a normal call to adios_write using the same io_handle, field_name, and the returned buffer.
Adios_start_calculation	An indicator that it is now an ideal time to do bulk data transfers as the code will not be performing IO for a while.
Adios_end_ calculation	An indicator that it is no longer a good time to do bulk data transfers as the code is about to start doing communication with other nodes causing possible conflicts
Adios_end_iteration	A tick counter for the IO routines to time how fast they are emptying the buffers.